



Dec. 91/Jan. 92
Vol. 1 No. 3

INSIDE ASSEMBLER™

Tips & techniques for Assembler • PC

*Important notice
for subscribers inside...*

LAST ISSUE!



A Publication of The Cobb Group

IMPORTANT MESSAGE

*Please read
this message!*


Dear Subscriber:

This is the final issue of *Inside Assembler*. But don't worry, I won't leave you empty-handed. I'm going to make up the balance of your subscription with *The DOS Authority*, our journal for advanced DOS users.

As a PC expert, I think you'll enjoy what *The DOS Authority* has to offer. Every month, you'll find timesaving shortcuts to help you use DOS more effectively. *The DOS Authority* will show you how to take advantage of DOS 5's shell, configure DOS for optimal performance, execute advanced batch file programming techniques, and much more.

If you don't wish to receive *The DOS Authority*, you can choose one of our other journals listed on the inside back page of this issue. Just call our Customer Relations Department at (800) 223-8720 or (502) 491-1900 to let us know which journal you'd like to receive instead.

Sincerely,



Douglas Cobb, Publisher
The Cobb Group

INSIDE ASSEMBLER™

Tips & techniques for Microsoft Assembler

Directly accessing the text mode video screen

By Marco C. Mason

Listing 1 begins on page 13

The IBM PC and its myriad compatibles have many options for controlling the video screen, ranging from 40x25 monochrome to high-resolution Professional Graphics Controller (PGC) and XGA color graphics. In this article, we'll focus on a subset of these modes: the text modes. First, we'll examine how the various text modes work. Then we'll show you how to bypass DOS and BIOS and directly access the screen for higher speed. Finally, we will present a simple program that shows you how to put it all together.

There are five standard text modes used by the IBM PC. Table A shows a list of these modes and the adapters that support them. Be aware that, while most IBM video controller cards support only a subset of these modes, some third-party cards can support all standard text modes. Also note that since IBM CGA adapters disable the colorburst signal in modes 0 and 2, they don't show color in these modes. Most clone CGA cards don't disable the colorburst signal, so they *do* show color in modes 0 and 2.

TABLE A

| Mode | Resolution | RAM address | Adapters |
|------|---------------|-------------|----------|
| 0 | 40x25 (mono) | b800:0000 | 1 |
| 1 | 40x25 (color) | b800:0000 | 1 |
| 2 | 80x25 (mono) | b800:0000 | 1 |
| 3 | 80x25 (color) | b800:0000 | 1 |
| 7 | 80x25 (mono) | b000:0000 | 2 |

| | | | |
|---|--|--|--|
| Adapter list: | | | |
| 1 - CGA, EGA, MCGA, VGA, and XGA | | | |
| 2 - MDA, VGA, XGA, and Hercules graphics card | | | |

Luckily, the architecture of the various text mode screens is extremely similar. This enables us to use nearly identical code when managing these different text mode screens.

How the memory is formatted

In text mode, fully representing a character on the screen takes two bytes. For each pair, the first byte is the code for the character and the second byte controls its attributes: the foreground color, background color, blink attribute, and intensity attribute.

The codes used to describe characters on the screen (called the IBM PC character set) are similar to the ASCII character set. The codes for the printable characters in the ASCII set (including codes 20H through 7EH, which represent numbers, letters, and punctuation symbols) are the same as those in the IBM PC character set. In addition, the IBM PC character set includes symbols for the unprintable characters in the ASCII set (characters 00H through 31H and 7FH) as well as the characters that are not defined by ASCII (80H through FFH).

The second byte in the definition of a character controls its display attributes. Table B on the next page shows the attributes controlled by each bit in the attribute byte, and Table C shows the different foreground and background colors for color video adapters (CGA through XGA).

While the effects of Bits 7 (blink) and 3 (intensity) are the same on all video adapters, the effects of Bits 6, 5, and 4 (the background color) and Bits 2, 1, and 0 (the foreground color) are different with monochrome adapters (MDA, Hercules). Table D shows the only four acceptable

Continued on page 2

IN THIS ISSUE

- Directly accessing the text mode video screen 1
- A collection of signed and unsigned decimal I/O functions 4
- Avoid being bitten by reserved words 9
- Forcing your computer to reboot 10
- Using input scripts with the CodeView debugger 11

INSIDE ASSEMBLER

Inside Assembler (ISSN 1057-560X) is published bi-monthly by The Cobb Group.

Prices Domestic\$59/yr. (\$14 each)
Outside US\$69/yr. (\$17 each)

Address The Cobb Group
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220

Phone Toll-free(800) 223-8720
(502) 491-1900
FAX(502) 491-4200

Staff Editor-in-ChiefMarco C. Mason
Publications MgrToni Bowers
EditingDuane Spurlock
Gregory L. Harris
ProductionGina Sledge
Eric Paul
DesignKarl Feige
PublisherDouglas Cobb

Address correspondence and special requests to The Editor, *Inside Assembler*, at the address above. Address subscriptions, fulfillment questions and requests for bulk orders to Customer Relations, at the address above.

Postmaster: Send address changes to *Inside Assembler*, P.O. Box 35160, Louisville, KY 40232. Second class postage is pending in Louisville, KY.

Copyright © 1991, The Cobb Group. All rights reserved. *Inside Assembler* is an independently produced publication of The Cobb Group. No part of this journal may be used or reproduced in any fashion (except in brief quotations used in critical articles and reviews) without prior consent of The Cobb Group.

The Cobb Group, its logo, and the Satisfaction Guaranteed statement and seal are registered trademarks of The Cobb Group. *Inside Assembler* is a trademark of The Cobb Group. Microsoft is a registered trademark of Microsoft Corporation. IBM is a registered trademark of International Business Machines, Inc.

STATEMENT OF OWNERSHIP, MANAGEMENT AND CIRCULATION
Statement of ownership, management and circulation (Required by 39 U.S.C. 3685). 1A. Title of publication: *Inside Assembler*. 1B. Publication No. 1057560X. 2. Date of filing: September 26, 1991. 3. Frequency of issue: Bi-Monthly. 3A. No. of issues published annually: 6. 3B. Annual subscription price: \$59/6 (\$69/6 foreign). 4. Complete mailing address of known office of publication: The Cobb Group, 9420 Bunsen Parkway, Suite 300, Louisville, Kentucky 40220. 5. Complete mailing address of headquarters of general business offices of the publisher: The Cobb Group, 9420 Bunsen Parkway, Suite 300, Louisville, Kentucky 40220. 6. Full names and complete mailing address of publisher, editor, and managing editor: Publisher, Douglas F. Cobb, The Cobb Group, 9420 Bunsen Parkway, Suite 300, Louisville, Kentucky 40220; Editor, Marco C. Mason, The Cobb Group, 9420 Bunsen Parkway, Suite 300, Louisville, Kentucky 40220. Managing Editor, Linda Baughman, The Cobb Group, 9420 Bunsen Parkway, Suite 300, Louisville, Kentucky 40220. 7. Owner: Ziff Communications Co., Ziff Investment Partnership L.P., Philip B. Korsant, the address for all of the foregoing being 1 Park Ave. New York, NY 10016. 8. Known bondholders, mortgagees, and other security holders owning or holding 1 percent or more of total amount of bonds, mortgages or other securities: None. 9. For completion by nonprofit organizations authorized to mail at special rates (DMM Section 423.12 only)—The purpose, function, and nonprofit status of this organization and the exempt status for Federal income tax purposes (Check one): ☐ Has not changed during preceding 12 months. ☐ Has changed during preceding 12 months. (If changed, publisher must submit explanation of change with this statement.) 10. Extent and nature of circulation—A. Total no. copies (net press run): Average no. copies each issue during preceding 12 months, 3,567; actual no. copies of single issue published nearest to filing date, 2,902. B. Paid and/or requested circulation—1. Sales through dealers and carriers, street vendors and counter sales: Average no. copies each issue during preceding 12 months, 10; actual no. copies of single issue published nearest to filing date, 11. 2. Mail subscription (Paid and/or requested): Average no. copies each issue during preceding 12 months, 1,555; actual no. copies of single issue published nearest to filing date, 1,555. C. Total paid and/or requested circulation: (Sum of 10B1 and 10B2) Average no. copies each issue during preceding 12 months, 1,565; actual no. copies of single issue published nearest to filing date, 1,566. D. Free distribution by mail, carrier or other means. Samples, complimentary, and other free copies: Average no. copies each issue during preceding 12 months, 16; actual no. copies of single issue published nearest to filing date, 16. E. Total distribution (Sum of C and D): Average no. copies each issue during preceding 12 months, 1,581; actual no. copies of single issue published nearest to filing date, 1,582. F. Copies not distributed. 1. Office use, left over, unaccounted, spoiled after printing: Average no. copies each issue during preceding 12 months, 1,986; actual no. copies of single issue published nearest to filing date, 1,320. 2. Return from news agents. Average no. copies each issue during preceding 12 months, 0; actual no. copies of single issue published nearest to filing date, 0. G. Total (Sum of E, F1 and 2—should equal net press run shown in A): Average no. copies each issue during preceding 12 months, 3,567; actual no. copies of single issue published nearest to filing date, 2,902. 11. I certify that the statements made by me above are correct and complete. Douglas F. Cobb, Publisher.

Continued from page 1

Directly accessing the text mode video screen

combinations of foreground and background color settings for monochrome adapters. When you write software that might be run on an MDA, make sure you give the user a method of setting the attributes to one of the legal values; otherwise, the screen could be illegible.

TABLE B

| Bit | Function |
|-------|-------------------------------------|
| 7 | Blink control (0=no blink, 1=blink) |
| 6...4 | Background color |
| 3 | Intensity (0=normal, 1=high) |
| 2...0 | Foreground color |

TABLE C

| Color | Bit pattern |
|------------|-------------|
| Black | 000 |
| Blue | 001 |
| Green | 010 |
| Cyan | 011 |
| Red | 100 |
| Magenta | 101 |
| Brown | 110 |
| Light Gray | 111 |

TABLE D

| Name | Background (Bits 6, 5, 4) | Foreground (Bits 2, 1, 0) |
|----------------------|---------------------------|---------------------------|
| Normal characters | 000 | 111 |
| Inverse video | 111 | 000 |
| Invisible characters | 000 | 000 |
| Underline | 000 | 001 |

Accessing the screen directly

Now that you know how your PC defines each character on the screen, it's easy to create a data structure you can use to represent a screen character. Figure A shows just such a structure.

Figure A

```
screenCell struc ; A definition of a screen cell
    char db ? ; the character on the screen
    attr db ? ; the visual attributes
screenCell ends
```

You can use the *screenCell* structure to access video memory.

Now that you know how the screen cell is defined, you can start accessing the video RAM directly. Armed with this knowledge, let's see how our demonstration program works.

VIDEO.ASM—an example program

The program VIDEO.ASM, shown in Listing 1A on page 13, demonstrates how you can directly access the text mode video RAM. Before we dive in and study its internal operation, here's a brief summary of how it works. First it clears the screen with an attribute of 00H. As you can see from Tables B, C, and D, this gives invisible characters on all standard video adapters. (Color adapters get black characters on a black background, and monochrome adapters get invisible characters.)

Next, VIDEO.ASM prints the message *Inside Assembler* down the center of the screen, on all lines but the first and last. Then the program waits for you to press a key. After you do so, it sets the screen attributes to gray characters on a black background. Then VIDEO.ASM waits for you to press another key. When you do, it changes the screen attributes to black characters on a white background. Finally, it waits for you to press one more key, and once you do, VIDEO.ASM exits.

As you can see by looking at the code, we've put all the hard work in the three procedures `clear_screen`, `display_string`, and `set_screen_attr`. Let's take a look at these procedures in detail to see how they work.

The `clear_screen` procedure

The `clear_screen` procedure clears the screen by setting the `char` field of each screen cell to a space, and the `attr` field to a value you specify. You tell the `clear_screen` procedure what attribute you want by passing its value in the AL register.

When you call `clear_screen`, it first pushes some registers onto the stack so it can restore them to their original values before it returns. Then `clear_screen` calls `set_ES_to_screen_start` to get the segment address of the screen. (If `set_ES_to_screen_start` fails, it returns with the Carry flag set, and `clear_screen` simply aborts.)

Figure B

```
mov cx, SCREEN_SIZE
xor di, di           ; Now ES:DI points to screen start
mov ah, al           ; AH = attribute
mov al, ' '          ; AL = character (space)
rep stosw            ; set entire screen to blanks
```

This code is the heart of the `clear_screen` procedure.

As you can see in Figure B, `clear_screen` next sets the CX register to hold the number of cells in the screen, and

the DI register to point to the start of video memory. Then `clear_screen` places the video attribute in AH and a space in AL. (We do this because when you write a word, it reverses the bytes—which makes the character first and the attribute second, just as in the `screenCell` structure.) Finally, `clear_screen` fills all the screen cells with the specified character and attribute.

We'll describe the `set_screen_attr` procedure next because it's so similar to the `clear_screen` procedure.

The `set_screen_attr` procedure

The `set_screen_attr` procedure is nearly the same as `clear_screen`. The only difference is that `clear_screen` sets all the characters on the screen to blanks, while `set_screen_attr` doesn't change the characters—it changes only the attributes on the screen.

Figure C shows the heart of the `set_screen_attr` procedure. Instead of using `rep stosw` to set the character *and* the attribute, we've had to construct a loop. The first thing we do in this loop is place the video attribute in the `attr` field of the cell that DI points to. Next, `set_screen_attr` points to the next cell by adding 2 to DI. Then we iterate the loop until we've set the video attribute for each screen cell.

Figure C

```
mov cx, SCREEN_SIZE
xor di, di           ; Now, ES:DI points to video RAM
ssaLoop:
IF @Version GT 600
    ASSUME di:ptr screenCell
ENDIF
mov es:[di].attr, al ; put char on display
add di, 2            ; skip to next cell address
loop ssaLoop         ; process next cell on screen
```

The heart of the `set_screen_attr` procedure is similar to that of `clear_screen`.

(We could replace the highlighted lines in Figure C with a `stosb` and an `inc di`, which would execute faster on 8086 and 8088 machines, but we wanted to show you how to access the screen cells with the `screenCell` structure.)

While the `set_screen_attr` procedure changes only the attributes on the screen, the `display_string` procedure changes only text and ignores the attributes.

The `display_string` procedure

The `display_string` procedure is the most complex of the three procedures. It displays a string on the screen at the desired location. To use it, you must pass it three parameters: the row on the screen (in DH), the column (in DL), and the address of the string to print (in DS:SI). You terminate the string with a byte of zeros so `display_string` knows when to stop.

Since `display_string` doesn't display text all over the screen, it has two jobs. First it must decide where to place the text, then it must actually put the text in video memory.

Figure D shows the code that `display_string` uses to compute the location in memory to place the text. Since DH holds the row number and DL holds the column of the first character of the string, $DH * SCREEN_WIDTH + DL$ is the number of the first cell. Each cell is two bytes long, so the offset address of the screen cell is $2 * (DH * SCREEN_WIDTH + DL)$. The code in Figure D computes this value and places it in BX.

Figure D

```
mov cl, SCREEN_WIDTH ; cells_in_row
xor ax, ax
mov al, dh            ; row_number
mul cl               ; AX=row_number * cells_in_row
add al, dl           ; AX=row_number*cells_in_row+column
adc ah, 0
add ax, ax           ; each cell is two bytes long
mov bx, ax           ; BX = offset of first character
```

display_string uses this code to compute the location in video memory to place the string.

Figure E shows the loop that `display_string` uses to place the text in the video memory. Just before the loop executes, we compute how many characters are left on the row (so we don't go past the left margin and wrap to the right of the screen).

First the loop loads the character into AL and tests it to see whether it's the end of the string value (i.e., if it's zero)—if it is, we exit the loop. If it doesn't signify the end of the string, we place the value in video memory and point to the next cell. Then we check to see if we've hit the right margin yet—if so, we exit the loop.

In all three of these procedures, we called a procedure named `set_ES_to_screen_start` to get the segment

address of the start of the video display memory. Since this program works only in the screen mode you select when you assemble it, it's a reasonable compromise. If you modify this program to work with multiple screen modes, you'll need to write something fancier than `set_ES_to_screen_start`.

Figure E

```
sub cl, dl           ; compute the max # of columns
ds_loop:
  lodsb              ; get the character to display
  or al, al          ; End of string found?
  jz ds_done
  IF @Version GT 600
    ASSUME bx:ptr screenCell
  ENDIF
  mov es:[bx].char, al ; put char on display
  add bx, 2           ; point to the next screen cell
  dec cl
  jnz ds_loop
ds_done:              ; done-EOString or RMargin
```

display_string uses this code to copy the string to video memory without disturbing the video attributes.

Conclusion

When you assemble the program, you need to supply the value of the `SCREENMODE` equate. You can do this on the command line with the `-D` switch. If you supply an invalid value for `SCREENMODE`, the program won't run properly. Additionally, when you run your program, the adapter must already be in the mode you specify when you assemble it—we don't change the video mode in this program.

Now that you know how the video screen memory is organized, you can read and write text to the screen in the blink of an eye. ■

INPUT AND OUTPUT ROUTINES

A collection of signed and unsigned decimal I/O functions

Listings 2A and 2B begin on page 14

In our first two issues, we discussed a set of hexadecimal input and output functions. While hexadecimal notation is great for computer programmers, it's not

quite what the rest of the world is used to. In this article, we'll discuss a small set of procedures designed to help you interact with users with signed and unsigned decimal

numbers. First we'll discuss how we built the output functions, then we'll look at the input functions.

The basic problems of formatting numbers for output

Unfortunately, hexadecimal I/O is much simpler than decimal I/O, since you're dealing with an even four bits at a time. Decimal numbers aren't as nice to work with in binary computers because you can't just extract the middle digit easily, as you can in hexadecimal.

There are three problems we'll have to overcome in order to output decimal numbers. First, we need to be able to extract the decimal digits from the binary number. Second, we need to format the number; third, we need to handle the sign for signed decimal numbers. As we solve these problems, we'll go ahead and build parts of our decimal number output library.

Let's start with the first problem: How do we extract a decimal digit from a binary number? Two solutions come to mind. We could successively subtract powers of ten until the number goes negative and keep track of how many subtractions we're made. Figure A shows some code that functions like this.

Figure A

```

mov cx, 10000    ; Let's find the 10,000's digit
xor bl, bl       ; so far we have no 10,000's
loop:
  sub ax, cx
  jc done        ; if AX < CX, we're done!
  inc bl         ; otherwise, add 1 to the 10,000's
count
  jmp loop       ; and try again
done:
  add ax, cx     ; we're done
                  ; so add 10,000 to make it positive
                  ; now we're ready to check 1,000's ...

```

This code can determine how many 10,000's exist in AX by using successive subtraction.

Thus, if you start with AX=35,437, you'll go through the loop as shown in Table A.

While you can code it better, the method in Figure A suffers from being slow. A better way is to divide the number in the AX register by 10. When you do this, it places the remainder in the DX register and the quotient in AX. Each time you divide the value in AX by 10, you get another digit. Figure B shows a fragment of code that successively divides the value in AX by 10.

TABLE A

| Value in AX | Value in BX | Description |
|-------------|-------------|---|
| 35437 | 0 | First time we hit label loop |
| 25437 | 1 | Second time |
| 15437 | 2 | Third time |
| 05437 | 3 | Fourth time |
| -4563 | 3 | Fifth time; AX<0, so we quit |
| 05437 | 3 | After the add ax, cx to fix the AX register |

Figure B

```

mov bx, 10 ; For decimal numbers, use divisor of 10
loop:
  xor dx, dx ; zero out the MSW of the dividend
  div ax, bx ; AX = AX/10, DX=remainder
  or ax, ax ; When AX=0, we've gotten all the digits
  jnz loop

```

This code fragment divides the value in AX by 10 successively until AX is zero.

Let's trace through the code in Figure B, again using 35,437 as our starting value in AX. We'll trace our progress in Table B.

TABLE B

| Value in AX | Value in DX | Description |
|-------------|-------------|--------------------------------|
| 35437 | ? | When we first hit loop |
| 3543 | 7 | Next iteration (units) |
| 354 | 3 | Next iteration (tens) |
| 35 | 4 | Next iteration (hundreds) |
| 3 | 5 | Next iteration (thousands) |
| 0 | 3 | Last iteration (ten thousands) |
| | | AX = 0, so we quit |

While the code isn't any more complex, we get a digit each iteration! (Of course, the DIV instruction is pretty slow, but it beats repeatedly executing the loop in Figure A for each digit.) The only potential problem is that we get the digits in reverse order.

The convert_bin_to_string procedure

It turns out that this second section of code is a critical component of our decimal output functions, so we built a

helper routine out of it called `convert_bin_to_string`, which some of our other functions will call to make a decimal string. (The code for `convert_bin_to_string` and the other procedures we discuss in this article is included in DECIO.ASM, found in Listing 2A on page 14.)

When you call `convert_bin_to_string`, you pass it the binary number in AX, and it will return a pointer to the string in the DS:SI register pair and the number of digits in the CL register.

Figure C shows the code for `convert_bin_to_string`. We only had to convert the value in DL to ASCII and store it into a buffer to end up with a useful function. We put our string in a temporary buffer named `temp_buff` because we need to do more work on the string of digits before we place it in the output buffer.

Figure C

```
convert_bin_to_string proc
    mov si, offset temp_buff+9 ; put str in temp buffer
    mov bx, 10
    xor dx, dx                ; clear top half of accumulator
    xor cl, cl                ; clear number of digits count
convert_loop:
    div bx                    ; get least significant digit
    add dl, '0'               ; turn into an ASCII digit
    dec si                    ; point to place to store digit
    mov [si], dl              ; store digit
    inc cl                    ; increment digit count
    xor dl, dl                ; clear top half of accumulator
    or ax, ax                 ; are we done yet?
    jnz convert_loop
    ret
convert_bin_to_string endp
```

Here's the complete `convert_bin_to_string` procedure, which we'll call in our other functions.

The major problem with `convert_bin_to_string` is that it's very difficult to print numbers in nice, pretty columns, as users expect. To combat this problem, we created another helper procedure named `move_to_output_buffer`.

The `move_to_output_buffer` procedure

Since you want your programs to have nice output, you often want to control the width of the number you're printing. The `move_to_output_buffer` procedure copies the decimal string from the temporary buffer to the output buffer, padding the string with blanks if necessary. If the number is too wide to fit in the field, then `move_to_output_buffer` copies the entire number into the output buffer. (`move_to_output_buffer` assumes that outputting the correct number is a higher priority than making your columns line up.)

`move_to_output_buffer` expects the DS:SI register pair to point to the work buffer, CL to hold the number of digits in

the work buffer, and ES:DI to point to the output buffer.

Figure D shows the code for the `move_to_output_buffer` procedure. First, it gets the desired `field_width` into the AL register. Then, if the value in the AL register is larger than the one in the CL register, it places AL-CH spaces in the output buffer. After padding the output field with blanks, `move_to_output_buffer` copies the work buffer to the output buffer.

Figure D

```
move_to_output_buffer proc
    ; pad output field with enough ' ' to fill field
    xor ch, ch
    push cx
    mov al, field_width      ; # to pad=field_width-num_dig
    sub al, cl
    jc done_padding          ; don't pad if not enough room
    xchg al, cl              ; CX = number of spaces to add
    mov al, ' '
    rep stosb                ; pad output buffer
done_padding:
    pop cx                  ; CX = # digits in temp buff
    rep movsb               ; copy digits to output buffer
    ret
move_to_output_buffer endp
```

The `move_to_output_buffer` procedure copies the number prepared by `convert_bin_to_string` to the output buffer, padding if necessary.

Now we've completed the basic building blocks for our output functions. Let's go ahead and build our signed and unsigned decimal output functions with them.

The `udec_format` procedure

Now that we've solved the first two problems, we can go ahead and create our first ready-to-use procedure: `udec_format`. You can call `udec_format` with a number in AX and the address of your output buffer in ES:DI, and it will format the number as a string of unsigned decimal digits. When it returns, ES:DI will point to the next available position in the buffer, and AX will contain garbage.

While the `udec_format` procedure doesn't require any further explanation (it relies solely on the procedures `convert_bin_to_string` and `move_to_output_buffer`), the next function, `dec_format`, does need a little more clarification.

The procedure `dec_format`

The only thing that complicates `dec_format` is the third problem we mentioned earlier—we have to manipulate the sign. So before we call the `convert_bin_to_string` procedure to convert the number to a string of digits, we need to take care of its sign.

Figure E contains the snippet of code that we use to take care of the number's sign. If the number is negative, we set the CH register to 1 and negate the number (turning it positive). If the number is positive, we set CH to 0. We'll use this CH register value to determine whether or not to add a '-' to the output buffer.

Figure E

```
xor ch, ch      ; sign flag: 0=positive, 1=negative
or ax, ax      ; is number < 0?
jns positive
neg ax         ; # is negative, make it positive
inc ch        ; and remember its sign
positive:
```

Here's the code dec_format uses to manipulate the number's sign.

Now that we've taken care of the sign, we can call convert_bin_to_string; when it returns, we can use the value in CH to decide whether or not to add a negative sign. Figure F shows the code we use to do this.

Figure F

```
or ch, ch      ; Is it a negative number?
jz positive2
dec si         ; Yes, make room for '-'
mov byte ptr [si], '-' ; add '-' to the buffer
inc cl        ; update the digit count
positive2:
```

dec_format uses this code to place the minus sign in the temporary work buffer for negative numbers.

After dec_format adds the sign to the temporary work buffer, it simply calls move_to_output_buffer to copy the string to the output buffer and finish the job.

Interpreting signed and unsigned decimal numbers

Reading and interpreting signed and unsigned decimal numbers is of similar complexity to outputting them. Again, we have to construct the binary number from the input buffer, and again we have to worry about the sign. For input, however, we don't have to worry about formatting the result; instead, we have to concern ourselves with input errors.

Let's concentrate on the easy part—building a binary number digit by digit. We'll worry about the errors as we come to them.

In the last section, we constructed a couple of building block functions before building the final functions.

We're going to do that again here—except that we need only one building block.

The procedure convert_string_to_bin

The method we'll use to convert a string to a binary number is simple: Each time we encounter a digit, we'll multiply our previous number by 10, and add the value of the current digit. We'll keep going until either we run out of digits or the number gets too big.

Figure G shows the code for our new building block function: convert_string_to_bin. If convert_string_to_bin fails (i.e., the number is too big), it returns with the Carry Flag set; otherwise, it returns with the Carry Flag clear.

Figure G

```
convert_string_to_bin proc
    xor dx, dx      ; zero accumulator
    xor ax, ax
    mov cx, 10      ; the number base
convert:
    mov bl, [di]    ; get a character
    sub bl, '0'     ; is it a digit?
    jb done
    cmp bl, 9
    ja done
    inc di          ; point to the next character
    mul cx          ; put digit in accumulator (i.e.,
    add al, bl      ; mult old value by 10 and add
    adc ah, 0       ; the new digit.)
    adc dx, 0
    jz convert      ; if no overflow, process next digit
    stc             ; overflow error! (# > 65536)
    ret
done:
    cld
    ret
convert_string_to_bin endp
```

Here's the core function that reads a string and returns a binary number in AX.

The udec_read procedure

As you may imagine, the udec_read procedure is pretty simple, given the function convert_string_to_bin. In fact, all we need to do is to call convert_string_to_bin and check for two user errors.

One error the user might make is to type a number that's too large. Luckily, convert_string_to_bin detects this error, and when it does, it returns with the Carry Flag set. The other possible error the user might commit is to call udec_read without a string of decimal digits. This error is just as easy to trap, since we just look at the buffer pointer before and after the call to convert_string_to_bin, and if it

doesn't change (i.e., we didn't process any characters), the user gave `udec_read` an invalid string.

When you call `udec_read` in your programs, pass it the address of the input buffer in `DS:DI`. `udec_read` returns the value (or the error code) in the `AX` register, with `DS:DI` pointing to the first unused character in the input buffer, and the Carry Flag set if there was an error. (If the Carry Flag is set, `AX` contains the error code `er$OVFLW` (2) if the number was too large and `er$BADCHR` (1) if `udec_read` couldn't process any characters.)

The `dec_read` procedure

The `dec_read` procedure is slightly more complex than `udec_read` because it must handle both positive and negative numbers. Since `convert_string_to_bin` recognizes only digits, it chokes if you pass it a '+' or '-' symbol. Therefore, the first thing that `dec_read` must do is check if there is a sign character on the front of the number, and if there is, remember it and remove it. The code we use to do this is shown in Figure H.

Figure H

```
xor bh, bh      ; default is no sign (BH=0=positive)
mov bl, [di]    ; get character
cmp bl, '+'     ; if it's a '+', skip and ignore it
jz dr_positive
cmp bl, '-'     ; if it's a '-', set BH to 1 (neg)
jnz dr_ignore
inc bh
dr_positive:
inc di
dr_ignore:
```

Here's the code that `dec_read` uses to process a leading sign, if one exists.

After `dec_read` processes the leading sign, it calls `convert_string_to_bin` to get the binary equivalent of the decimal string. When `convert_string_to_bin` returns, `dec_read` has to adjust the sign and detect errors. Figure I shows the code `dec_read` uses to process the sign and check for input errors.

The error handling and sign processing are inextricably intermingled in the code in Figure I. Here's a brief synopsis: First we check if the number should be negative (i.e., if it was preceded by a '-'). If not, then we check to see whether it's negative anyway (because values from 32,768 to 65,535 didn't overflow in the `convert_string_to_bin` function, but they're interpreted as negative values in `dec_read`). If so, we issue an overflow error and quit. If the number isn't negative, and we processed digits, then we exit cleanly with the value in `AX`.

Figure I

```
call convert_string_to_bin
jc dr_error1      ; if # is too large, overflow
or bh, bh        ; was '-' at beginning of number?
jnz dr_negative
or ax, ax        ; is number in AX negative?
js dr_error1      ; (it shouldn't be, yet!)
cmp si, di       ; did we process any digits?
jz dr_error2

dr_exit:          ; NORMAL EXIT
clc              ; indicate no error

dr_rejoin:
; pops to restore registers
ret

dr_negative:      ; number was preceded by '-' sign
inc si           ; did we process any digits?
cmp si, di
jz dr_error2
or ax, ax        ; Is the number 0?
jz dr_exit
neg ax           ; Negate the number
jns dr_error1    ; If not negative, we overflowed
jmp dr_exit

dr_error1:        ; ABNORMAL EXIT
stc              ; indicate an error occurred
mov ax, er$OVFLW ; error number == OVERFLOW
jmp dr_rejoin

dr_error2:        ; ABNORMAL EXIT
stc              ; indicate an error occurred
mov ax, er$BADCHR ; error number == BAD CHARACTER
jmp dr_rejoin
```

The error checking for `dec_read` is much more complex than that required by `udec_read`.

If the number is intended to be negative, then we check to see if we processed any digits. If not, we overflow with a bad input character error. Next we have to check if the number is zero (because our next test fails for zero). If it's zero, then we exit cleanly. Next we negate the number, and if it's not negative we exit with an overflow error.

The `dec_set_width` procedure

The only procedure we didn't discuss yet is `dec_set_width`. It really doesn't do anything special—when you pass it a field width in the `AL` register, it sets the `field_width` variable to that value. When `dec_set_width` returns, it returns the previous value of `field_width` in the `AL` register.

DECTST.ASM—the example program

DECTST.ASM, shown in Listing 2B on page 16, simply gives the decimal I/O routines a workout. If you define SIGNED when you assemble it, DECTST.ASM tests the signed decimal I/O; and if you assemble it without defining SIGNED, it tests the unsigned decimal I/O functions.

There's nothing out of the ordinary in DECTST.ASM, so here's a thumbnail sketch of its operation: It prompts

you for two numbers separated by a space (though if you enter more than one space, DECTST will skip over them properly). Then it parses both the numbers and adds them. Finally, it prints the sum of the two numbers.

Conclusion

Now you can use decimal I/O in your programs so other people won't have to wonder what kind of number 38A5 is. They'll be able to interact with your programs on a meaningful level. ■

BEWARE OF THIS TRAP!

Avoid being bitten by reserved words

When you're programming in Microsoft Assembler 6.0, you'll frequently encounter a problem—running into reserved words. MASM 6.0 is very powerful, but it incurs much of this power at the expense of a cluttered name space.

If you envision a large room as the “space” that you can put names in, you want the manufacturer of an assembler to put reserved words out of your way—i.e., near the walls. Unfortunately, in assembly language, the opcodes are reserved words, the directives are reserved words...there are reserved words all over the place. It's much like having a bunch of marbles scattered all over the floor. When you walk through the room, you're always stepping on marbles. Programming in assembly language is similar to walking through this room.

For example, you might think that `loop` is a good label for the start of a loop. Unfortunately, it's also the name of an instruction. If you forget that `loop` is an instruction (and hence, a reserved word), then when you use it as a label you'll get the following error:

```
ERRORS.ASM(15): error A2008: syntax error : loop
```

MASM won't tell you when you use a reserved word in an inappropriate context. Instead, it will give you some oddball error message that often obscures the true cause of the error.

In languages like Pascal or C, the problem still exists, but less so—there are so few reserved words in these two languages. MASM 6.0, however, has more than 600 reserved words. This means that just about any short, mnemonic name you might think of to use in a program has probably already been reserved, and you can't use it.

How to avoid the reserved word problem

Luckily, there are two easy ways to minimize the reserved word problem. The first method is to use long variable names,

which are preferable to short variable names because most of the reserved words are three or four characters long. An added benefit of using long variable names is that your programs become easier to understand—assuming your variable names mean something.

For example, the program in Figure A tries to print a string named `str` to the screen. Unfortunately, `str` is the mnemonic for the *Store Task Register* command on the 80286 (and later) CPUs.

Figure A

```
.model small
.stack
.data
str db "This is a string$"
.code
.286
.startup
    mov dx, offset str
    mov ah, 9
    int 21h
.exit 0
end
```

This program fails because a data item has the same name as a reserved word.

When you assemble this program, you'll get the following error messages:

```
ERROR1.asm(4): error A2085: instruction or register not
accepted in current CPU mode
ERROR1.asm(8): error A2008: syntax error : str
```

As you can see, the first occurrence of `str` causes MASM to issue the A2085 error, and the second

occurrence causes MASM to issue the A2008 error. The error messages are different because they appear in sections with different CPU levels. The first error occurs in standard 8086 mode, while the second occurs in 80286 mode.

Since the 8086 CPU doesn't have the `str` instruction, MASM warns you that the instruction is invalid in the current mode for the first occurrence. In the second instance, since MASM is in 80286 mode, it supports the instruction—but it detects that you've used it incorrectly. That's why you get two different error messages. You can fix both errors selecting a better name for your string.

You can employ another method to avoid the reserved word trap—use special characters in your names. The special characters (`@`, `_`, `$`, and `?`) can also help because few of the reserved words use them. (Microsoft begins some reserved words with `@`, so you should avoid this character.)

Conclusion

MASM 6.0's reserved words can be a real bugaboo when you're programming because the error messages can be misleading. Consequently, you can spend too much time chasing your tail looking for nonexistent bugs. Luckily, using longer names and adding special characters to your names can make this a rare bug, indeed. ■

A SIMPLE UTILITY PROGRAM

Forcing your computer to reboot

Yes, I know—when you program in assembly language, you often reboot your computer—usually when you don't mean to. Still, it's occasionally convenient to force your computer to reboot. For example, you might write a program that changes your `CONFIG.SYS` or `AUTOEXEC.BAT` file, and to make the changes take effect, you must reboot your computer. In this article, we'll show you a technique you can use to reboot your computer.

The two types of booting

You can boot your computer in two ways. When you turn on the power or push the reset button (provided by some PC-compatible computers), you're performing a cold boot. When you reboot from the keyboard (with `Ctrl-Alt-Delete`), you're performing a warm boot.

In a cold boot, your computer starts over as if you had just turned on the power. The computer tests itself thoroughly, and loads DOS. A warm boot is similar, but the computer doesn't test itself as thoroughly when it starts up—it has already booted up successfully, so there's no need to test the computer as rigorously.

Your computer differentiates between a cold boot and a warm boot by examining a special memory location in the BIOS data area called, in some sources, the POST reset flag. (POST is an acronym for "Power On Self Test.") If this word of RAM holds 1234H, the computer performs a warm boot; otherwise, it performs a cold boot. (In fact, there are several other values of the POST reset flag that have different meanings on some computers, but they aren't consistent on all machines.)

How to reboot your computer

Since we want to write a program that can reboot the computer, we need to know the methods available to us.

Unfortunately, many references recommend using `INT 19H`—unfortunate in that it works reliably only on IBM computers. Most PC clones don't support the `INT 19H` method of rebooting.

Luckily, every computer we've tried supports the alternate method: performing a direct jump to the CPU restart address. The 80x86 series of processors all jump to address `0F000:FFF0H` whenever they're reset. Since this address is in ROM, it will work with your computer (unless you have a memory manager that swaps out your BIOS ROM).

Since the ultimate goal is to write a program to reboot your computer, it's obvious that your computer must boot successfully before it can run the program, so the program can use a warm boot to save time. Figure A contains the program we wrote that you can use to reboot your computer.

In order to specify the type of boot we want to perform, `REBOOT.ASM` must access the POST reset flag. To do this, we first put the address of the BIOS data segment (`0040H`) into the `ES` register. Next we place the offset address of the POST reset flag (`0072H`) into the `BX` register. The first three `mov` lines do this job—now `REBOOT.ASM` can access the POST reset flag.

The next two lines store 1234H into the POST reset flag to specify a warm boot. If you would rather have `REBOOT.ASM` perform a cold boot, you can replace the line

```
mov ax, 01234h
```

with the line

```
xor ax, ax
```

which causes `REBOOT.ASM` to store a value of 0 in the POST reset flag.

Figure A

```
*****
; REBOOT.ASM - a program to reboot your computer
;
; To compile: ml /AT reboot.asm
*****

.model tiny
.code
    org 0100h

reboot:
    mov ax, 0040h    ; Point to BIOS data
    mov es, ax       ; segment
    mov bx, 0072h    ; Point to POST reset flag
    mov ax, 01234h   ; Signal a WARM boot
    mov es:[bx], ax
    db 0eah          ; Jump to 0F000:0FFF0H
    dd 0f000fff0h

end reboot
```

This simple program reboots your computer.

Finally, REBOOT.ASM jumps to address 0F000:0FFF0H, which is where the CPU starts whenever you turn it on. We couldn't use the following line of code because Microsoft Assembler doesn't understand this instruction.

```
jmp 0f000fff0h
```

Instead, we created the instruction by hand—the opcode for a long jump is 0EAH and the address we want to jump to is 0F000:0FFF0H. You can make a macro using this technique to make an intersegment jump instructions.

While there is a better method we could use to perform a far jump to an absolute address, it doesn't work in *all* memory models. We devised this technique because it works everywhere.

Why did we select a method that works in all memory models when we're using the Tiny memory model? Because we want to be able to use the section of code printed in color in any program without having to modify it.

You'll notice we printed the section of code labeled *reboot* in blue. This is the only section you need to put in your programs in order to reboot them. The rest is just enough window dressing to get MASM to compile it into an EXE file.

Summary

Now you know which rebooting method is best (the jump to 0F000:FFF0H) and why (it works properly on more computers). We've also presented a subroutine you can use in any program in order to reboot your computer whenever the need should arise. ■

SAVE TIME AND EFFORT

Using input scripts with the CodeView debugger

How often have you debugged the same program? Well, unless you don't write programs, you probably spend a good deal of time debugging. In this article, you'll see a technique you can use in CodeView to minimize the amount of time you spend in the debugger.

We're going to show you what input scripts are, how to use them, describe how to create them, and (most importantly) explain why you want to use them. Obviously, the first question is, What *is* an input script?

What is an input script?

Simply put, an input script is a file containing a list of commands for CodeView to execute. The file is a simple

text file you can create with a text editor, like EDLIN. In this text file, not only can you use commands for CodeView, but you can add commands to help you figure out what CodeView is doing.

How to create an input script

While you can use EDLIN or another text editor to create an input script by hand, there's a better way. You can use the output redirection (T>) command to log what you're doing in CodeView. Then, when you're done with your debugging session, you have a complete transcript of what you typed and CodeView's responses. You can then use a text editor to clean up the log file and turn it into an input script.

In a long debugging session, you're not likely to remember the reason for including each thing you type, so you'll probably want to use the comment command (*) to document your work. Then, when you edit your debug log and turn it into an input script, you'll know what you did and why.

Working with input scripts can be frustrating at first, because if you notice a mistake in the script, you won't have a chance to correct anything until the script has run its course. Sometimes you'll get long streams of error messages from CodeView, and occasionally you'll lock up your machine. For this reason, you want to be particularly careful when you create the input script.

When you create an input script, you're likely to make one of two errors. The first error occurs when you include extra characters or omit necessary ones. The second error is the use of an incorrect address. Since CodeView accepts script commands as if you'd typed them, you need to be sure you enter them correctly.

For example, the register command (R) works three different ways, depending on the arguments you give it. If you give it no arguments at all, it simply displays the contents of the registers and tells you the current instruction. If you specify a register, it prints the register's current contents, then prompts you for a new value. If you specify both a register and a value, it sets the register to the new value, but prints nothing.

Therefore, if you used RAX to display the AX register, you must remember to follow it with a blank line so you'll leave the AX register unchanged. Otherwise, CodeView will consider the next command as the value you're trying to put in the AX register. Not only will CodeView probably interpret the command as an error, but in either case CodeView will ignore that instruction, causing your input script to fail.

Whenever you recompile your programs, memory addresses are liable to change. This can cause your script to fail if you use hardcoded addresses for breakpoints and memory areas. If you need to set a breakpoint, the safest way is to set it at a label. Whenever you compile your program, the address of the label may change, but CodeView will be aware of it.

How to use an input script

As we mentioned earlier, there are several commands that you can use to make using input scripts easier. First let's discuss three that you use *in* your input scripts: the comment (*), delay (:), and pause (") commands.

When a line starts with the comment command, CodeView simply displays the line in the command window and ignores the command. You can use comments to print messages on the screen informing you of what the script is about to do (or has already done).

Starting a line with a colon tells CodeView to wait for approximately one-half of a second before reading the next command. (You can use multiple colons on one line to

provide longer delays—each colon causes a one-half-second delay.) The delay command is most useful in a loop when you don't necessarily want to see every iteration. This lets you walk away from the computer while the program is running.

If you need longer delays, or you need to ensure that you can examine the screen in its entirety before processing the next command, you can use the pause command. When you issue this command, CodeView waits until you press a key on the keyboard before it continues processing characters. This way, you won't miss an important data item when you get up for a soda.

Since we described the technique of using CodeView's output redirection command to help build input scripts, you've probably guessed you can use CodeView's input redirection command (<) to run the script. In fact, that's exactly what you do. You can tell CodeView to read a list of commands to execute at any time—just follow the input redirection command with the name of the file holding the input script.

Why should you use input scripts?

Notice that we've left the most important question unanswered—*Why* should you use input scripts? You can think of input scripts as subroutines or macros for debugging. They make your job easier just as procedures and macros do in your programming. Perhaps this story will help you understand.

You have a complex function that computes your income tax (now *that* would be a complex function!). In order to test the function, you'll have to try it with many different inputs. Unfortunately, every time you run it, you may want to set up CodeView's windows, initialize some data areas, set a few watch variables, and run through the program until your function starts.

Each time you change your function, you have to recompile and re-examine the results in CodeView. If you have to type the following list of commands every time you debug your program, you're bound to make mistakes occasionally:

```
rax
0400
rsi
5510
es table1
2500 13.95 62.46
el table2
w?totalTax
w?checkToIRS
1.00001 16.4
g computetax
```

If you created a file with this list of commands, however, you could execute them all in one statement—

CodeView would automatically set things up for you *the same way* each time.

Conclusion

The input scripts technique for CodeView can make debugging a less tedious and tiresome job. When you

debug a program, you often have to perform the same operations over and over. Input scripts can let you build "super commands" for CodeView, so you can think in a higher-level point of view, rather than having to concentrate on myriad details. In this respect, they function like macros for MASM.

Source code listings *(You can download the listings from CGIS.)*

The following are the complete programs described in the preceding articles. We placed the listings at the end to preserve continuity, and we restricted them to 76 columns to maintain readability in the two-column format. You can download the listings from our online service, CGIS, at any

Listing 1: Directly accessing the text mode video screen

```

;*****
; VIDEO.ASM - demonstrate a way to access the text mode screen
; directly. YOU MUST SUPPLY THE VALUE FOR SCREENMODE BEFORE ASSEMBLY!
;*****
; To assemble <v6.0> ml /DSCREENMODE=?? video;
; <v5.1> masm -DSCREENMODE=?? video;
; link video;
;*****
; Where ?? = 0 or 1 for CGA/EGA/MCGA/VGA/XCGA screens in 40x25 mode,
;           2 or 3 for CGA/EGA/MCGA/VGA/XCGA screens in 80x25 mode,
;           7 for MDA/Hercules screens in 80x25 mode
;*****

.model small
.stack
.data

screenCell struc    ; Definition of the character on the screen
    char    db ?    ; The character to display
    attr    db ?    ; The attributes to use
screenCell ends

InsideAssembler db "Inside Assembler", 0

IF (SCREENMODE EQ 0) OR (SCREENMODE EQ 1)
    SCREEN_SIZE equ 40*25
    SCREEN_WIDTH equ 40
ELSE
    SCREEN_SIZE equ 80*25
    SCREEN_WIDTH equ 80
ENDIF

.code
start:
    mov dx, DGROUP    ; set DS and ES to point to the data
    mov es, dx        ; segment
    mov ds, dx
    cld                ; make sure DIRECTION = FORWARD

    ; Clear the screen
    mov al, 0          ; BLK on BLK - makes screen invisible
    call clear_screen
    jc abort

    ; Write message on screen
    mov dh, 23
    IF (SCREENMODE EQ 0) OR (SCREENMODE EQ 1)
        mov dl, (40 - 16) SHR 1
    ELSE
        mov dl, (80 - 16) SHR 1
    ENDIF
    mov si, offset InsideAssembler

mainLoop:
    call display_string
    jc abort
    dec dh
    jnz mainLoop
    xor ah, ah          ; get a key from the keyboard
    int 16h

    ; Set screen to gray letters (modes 0..3, white in 7) on black
    mov al, 07h
    call set_screen_attr
    jc abort
    xor ah, ah          ; get another key from keyboard
    int 16h

    ; Set screen to reverse video

```

time of the day. Use your 300-, 1200-, or 2400-baud modem with 8 data bits, 1 stop bit, and no parity to call CGIS at (502) 499-2904. Your user ID is your customer number (the one- to seven-digit number on the top line of your mailing label after the C:).

```

    mov al, 70h
    call set_screen_attr
    jc abort

    xor ah, ah          ; get another key from keyboard
    int 16h

abort:
    mov ax, 4c00h        ; quit program
    int 21h

;*****
; clear_screen - clear the screen (i.e., set the data and attribs for
; the entire screen).
;*****
; INP:  AL - screen attributes to use
;       OUT:  CY - SET if error (invalid screen mode), CLEAR otherwise
;       NOTE: destroys the AX register
;*****
clear_screen proc
    push cx                ; preserve the registers
    push di
    push es

    call set_ES_to_screen_start
    jc cs_error
    ; Set the attributes on the screen
    mov cx, SCREEN_SIZE    ; # of cells on screen
    xor di, di              ; start at 0, 0
    mov ah, al              ; Attribute
    mov al, ' '             ; Data
    rep stosw               ; set attrib & data for screen
    cld                     ; Done ... set NO ERROR flag

cs_error:    ; (note: if jc cs_error, then carry is set)
    pop es
    pop di
    pop cx
    ret
clear_screen endp

;*****
; display_string - put a message on the screen
;*****
; INP:  DH - row on screen to put message
;       DL - column on screen
;       DS:SI - points to string to display (terminated with 0)
;*****
display_string proc
    push ax                ; save the registers
    push bx
    push cx
    push si

    ; compute the starting location on the screen
    mov cl, SCREEN_WIDTH    ; cells_in_row
    xor ax, ax
    mov al, dh              ; row_number
    mul cl                  ; AX = row_number * cells_in_row
    add al, dl               ; AX = row_number*cells_in_row+column
    adc ah, 0
    add ax, ax               ; each cell is two bytes long
    mov bx, ax              ; BX = offset of first character

    ; set the ES register to the start of the screen
    call set_ES_to_screen_start
    sub cl, dl              ; Compute max number of columns

ds_loop:
    lodsb                  ; Get character to display
    or al, al               ; End of string found?
    jz ds_done
    jz ds_done
    IF @Version GT 600

```

```

        ASSUME bx:ptr screenCell
    ENDIF
    mov es:[bx].char, al      ; Put char on display
    add bx, 2                ; point to the next screen cell
    dec cl
    jnz ds_loop
ds_done:                ; done - either out of char or hit RM

    pop si
    pop cx
    pop bx
    pop ax
    ret
display_string endp

```

```

; set_screen_attr - set the entire screen to the specified attribute
;
; INP: AL - the screen attribute to use
; OUT: CY - Set if error (i.e., bad screen mode), otherwise Clear

```

```

set_screen_attr proc
    push cx                ; preserve the registers
    push di
    push es

```

```

    call set_ES_to_screen_start
    jc ssaExit

    ; Set the attributes on the screen
    mov cx, SCREEN_SIZE    ; # of cells on screen
    xor di, di              ; start at 0, 0

```

```

ssaLoop:
    IF @Version GT 600
        ASSUME di:ptr screenCell
    ENDIF
    mov es:[di].attr, al    ; Put char on display
    add di, 2               ; point to the next screen cell
    loop ssaLoop            ; process next cell on screen
    cld                     ; Done ... set NO ERROR flag

```

```

; Normal AND error exit point (when error detected, carry is set)
ssaExit:                ; NORMAL EXIT
    pop es
    pop di
    pop cx
    ret

```

```

set_screen_attr endp

```

```

; set_ES_to_screen_start - set the ES register to the segment that
; holds the screen data for the current video mode

```

```

; INP: - none -
; OUT: ES - holds 0B000H for mode 7 (mono), 0B800H for modes 0..3
;      CY - SET if invalid screen mode, CLEAR otherwise

```

```

set_ES_to_screen_start proc
    push cx                ; preserve cx
    mov cx, es             ; On error, don't change ES
    stc                    ; Error until otherwise proven
    IF SCREENMODE EQ 7
        mov cx, 0b000h     ; mono screen starts at b000h
        cld
    ENDIF
    IF (SCREENMODE GE 0) AND (SCREENMODE LE 3)
        mov cx, 0b800h     ; color screen starts at b800h
        cld
    ENDIF
    mov es, cx
    pop cx
    ret

```

```

set_ES_to_screen_start endp

```

```

end start

```

```

;-----
; dec_set_width - set the width of a decimal number field
;
; INP: AL - field width, where 0=automatically sized, and 1..255
;      are the legal widths
; OUT: AL - holds the last field width
;      CY - set if illegal field width, clear otherwise
;-----

```

```

public dec_set_width
dec_set_width proc

```

```

    xchg al, field_width    ; set field width & return old one
    ret

```

```

dec_set_width endp

```

```

;-----
; dec_format - format a number into a signed decimal string
;-----

```

```

; INP: AX - number to convert to a signed decimal string
;      ES:DI - address to place the resulting string
; OUT: AX - destroyed
;      ES:DI - points to next available place in buffer

```

```

public dec_format
dec_format proc

```

```

    push bx
    push cx
    push dx
    push si

```

```

    ; if the number is negative, put the sign in the buffer and
    ; negate it

```

```

    xor ch, ch              ; sign flag: 0=positive, 1=negative
    or ax, ax
    jns positive
    neg ax                  ; # is negative, make it positive
    inc ch                  ; and remember its sign

```

```

positive:

```

```

    call convert_bin_to_string ; convert AX to string in temp buffer

```

```

    ; Now put in sign (if appropriate)

```

```

    or ch, ch
    jz positive2
    dec si                  ; point to slot to hold '-'
    mov byte ptr [si], '-' ; store the '-'
    inc cl                  ; increment the digit count

```

```

positive2:

```

```

    call move_to_output_buffer ; Move formatted string to output buf

```

```

    pop si
    pop dx
    pop cx
    pop bx
    pop si
    ret

```

```

dec_format endp

```

```

;-----
; dec_read - convert a string of decimal digits to a binary number
;-----

```

```

; INP: DI - address holding the decimal number
; OUT: AX - decimal number (or error code if error)
;      DI - points to next character in buffer
;      CY - set on error, clear otherwise
; NOTE: the only error recognized is overflow (i.e. the user entered
;       too many digits)

```

```

public dec_read
dec_read proc

```

```

    push bx
    push cx
    push dx
    push si

```

```

    ; initialization

```

```

    mov si, di              ; keep track of first digit

```

```

    ; check for a sign
    xor bh, bh              ; default is no sign (BH=0=positive)
    mov bl, [di]            ; get character
    cmp bl, '+'              ; if it's a '+', skip and ignore it
    jz dr_positive
    cmp bl, '-'              ; if it's a '-', set BH to 1 (neg)
    jnz dr_ignore
    inc bh

```

```

dr_positive:

```

```

    inc di

```

```

dr_ignore:

```

```

    call convert_string_to_bin ; convert decimal string to binary
    jc dr_error1
    or bh, bh                ; Was '-' at beginning of number?

```

Listing 2A: A collection of signed and unsigned decimal...

```

;-----
; DECIO.ASM - signed and unsigned decimal I/O routines
;-----

```

```

;---Error codes
err$BADCHR equ 1          ; Bad character found in input string
err$OVFLW equ 2           ; Overflow -- number too large

```

```

.model small
.stack
.data

```

```

;---Data definitions
field_width db 0          ; field width (0=autosizing)
temp_buff db 10 dup(?)    ; temporary work buffer

```

```

.code

```



```

    jnz dr_negative          ; Is number in AX negative?
    or ax, ax                ; (it shouldn't be!)
    js dr_error1             ; Did we process any digits?
    cmp si, di
    jz dr_error2

dr_exit:                    ; NORMAL EXIT
    cld                     ; indicate no error

dr_rejoin:
    pop si
    pop dx
    pop cx
    pop bx
    ret

dr_negative:                ; Number was preceded by '-' sign.
    inc si                  ; Check if we've processed digits
    cmp si, di              ; other than the '-'
    jz dr_error2
    or ax, ax               ; Is the number 0?
    jz dr_exit
    neg ax                  ; Negate the number
    js dr_error1            ; If negating number doesn't make it
                           ; negative, we've overflowed ...

    jmp dr_exit

dr_error1:                  ; ABNORMAL EXIT
    stc                     ; indicate an error occurred
    mov ax, es$OVFLW        ; error number == OVERFLOW
    jmp dr_rejoin

dr_error2:                  ; ABNORMAL EXIT
    stc                     ; indicate an error occurred
    mov ax, es$BADCHR        ; error number == BAD CHARACTER
    jmp dr_rejoin

dec_read endp

;-----
; udec_format - format a number to an unsigned decimal string
;
; INP:  AX - number to convert to an unsigned decimal string
;       ES:DI - address to place the resulting string
; OUT:  AX - destroyed
;       ES:DI - points to next available place in buffer
;
public udec_format
udec_format proc
    push bx
    push cx
    push dx
    push si
    call convert_bin_to_string ; convert AX to string in temp buffer
    call move_to_output_buffer ; Copy the formatted string to the
                           ; output buffer

    pop si
    pop dx
    pop cx
    pop bx
    ret
udec_format endp

;-----
; udec_read - convert a string of decimal digits to a binary number
;
; INP:  DI - address holding the unsigned decimal string
; OUT:  AX - decimal number (or error code if error)
;       DI - points to next character in buffer
;       CY - set on error, clear otherwise
; NOTE: the only error recognized is overflow (i.e. the user entered
;       too many digits)
;
public udec_read
udec_read proc
    push bx
    push cx
    push dx
    push si

    mov si, di              ; remember first character position
    call convert_string_to_bin ; convert string to binary
    jc ur_error1

    ; NORMAL EXIT
    cmp si, di              ; If we didn't process any characters
    jz ur_error2            ; then we encountered a bad one
    cld                     ; no error ...

ur_rejoin:
    pop si
    pop dx
    pop cx
    pop bx
    ret

ur_error1:                  ; ABNORMAL EXIT - overflow error
    stc
    mov ax, es$OVFLW
    jmp ur_rejoin

```

```

ur_error2:                  ; ABNORMAL EXIT - bad character
    stc
    mov ax, es$BADCHR
    jmp ur_rejoin

```

udec_read endp

; move_to_output_buffer - a helper routine that moves the formatted
; number from the temporary work buffer to the output buffer, padding
; with spaces, if necessary.

```

; INP:  ES:DI - points to output buffer
;       DS:SI - points to string in temp work buffer
; OUT:  ES:DI - points to next available spot in the output buffer
; NOTE: destroys AL, CH, and SI

```

```

move_to_output_buffer proc
    ; pad output field with ' ' if not enough digits to fill field
    xor ch, ch
    push cx
    mov al, field_width      ; # to pad = field_width - num_dig
    sub al, cl
    jc done_padding         ; don't pad if not enough room
    xchg al, cl              ; CX = number of spaces to add
    mov al, ' '
    rep stosb               ; pad output buffer
done_padding:
    pop cx                  ; CX = number digits in temp buffer
    rep movsb               ; copy digits to output buffer
    ret

```

move_to_output_buffer endp

; convert_string_to_bin - read a decimal number from the string

```

; INP:  DS:DI - points to input buffer
; OUT:  AX - holds decimal number
;       DI - points to first character that's not a digit
;       CF - Set on overflow (unsigned number > 65536)
; NOTE: destroys BL, CX, DX

```

convert_string_to_bin proc

```

    xor dx, dx              ; zero accumulator
    xor ax, ax
    mov cx, 10              ; set base

convert:
    mov bl, [di]            ; get a character
    sub bl, '0'             ; is it in [0..9]?
    jb done
    cmp bl, 9
    ja done
    inc di                  ; point to next character
    mul cx                  ; put digit in accumulator (i.e.,
    add al, bl              ; mult old value by 10 and add
    adc ah, 0               ; the new digit)
    adc dx, 0
    jz convert              ; if no overflow, check next digit
    stc
    ret

```

done:

```

    cld
    ret

```

convert_string_to_bin endp

; convert_bin_to_string - convert a binary number to a string

```

; INP:  AX - holds the binary number
; OUT:  DS:SI - points to first character in formatted string
;       CL - holds length of the formatted string
; NOTE: destroys AX, BX, DX
; NOTE: procedure builds string from back to front

```

convert_bin_to_string proc

```

    mov si, offset temp_buf+9 ; end of buffer
    mov bx, 10                ; numeric base
    xor dx, dx                ; clear top half of the accumulator
    xor cl, cl

```

convert_loop:

```

    div bx                    ; Divide by 10 puts remainder in DL
    add dl, '0'               ; Convert remainder to ASCII digit
    dec si                    ; Point to the slot to hold digit
    mov [si], dl              ; Store ASCII digit into the buffer
    inc cl                    ; Keep track of # of digits used
    xor dl, dl                ; Clear remainder for next division
    or ax, ax                 ; check if we're done yet
    jnz convert_loop
    ret

```

convert_bin_to_string endp

end

Please include account number from label with any correspondence.

Listing 2B

```

;-----
; DECTST.ASM - exercise the decimal I/O routines
;
; To assemble...
; ...to test SIGNED <v6.0>: ml /DSIGNED dectst.asm decio.asm
; <v5.1>: masm -DSIGNED dectst;
; masm decio;
; link dectst+decio;
;
; ...to test UNSIGNED <v6.0>: ml dectst.asm decio.asm
; <v5.1>: masm dectst;
; masm decio;
; link dectst+decio;
;-----
.model small

extrn dec_read : proc, dec_format : proc, dec_set_width : proc
extrn udec_read : proc, udec_format : proc

;---Error codes
err$BADCHR equ 1 ; Bad character found in input string
err$OVRFLW equ 2 ; Overflow -- number too large

CR equ 0dh ; Carriage return
LF equ 0ah ; Line feed

.stack
.data

outBuf db 50 dup(0)
inpBuf db 15, 0, 15 dup(0)
Prompt db 'Please enter two decimal numbers separated by a '
       db 'space (or X to quit)', CR, LF, '$'

BadChar db CR, LF, 'Bad character in an input number. Please try '
        db 'again using only', CR, LF, 'the digits 0 through '
        db '9'
        IFDEF SIGNED
        db ' and + and - sign'
        ENDIF
        db CR, LF, '$'

TooLarge db CR, LF, 'You entered a number that was out of range. '
        db CR, LF, 'Please try again with a value between '
        IFDEF SIGNED
        db '-32768 and 32767', CR, LF, '$'
        ELSE
        db '0 and 65535', CR, LF, '$'
        ENDIF

Overflow db CR, LF, 'The two numbers added together are outside '
        db 'the normal range for', CR, LF
        IFDEF SIGNED
        db 'a SIGNED number (-32768 to 32767)'
        ELSE
        db 'an UNSIGNED number (0 to 65535)'
        ENDIF
        db CR, LF, '$'

;-----
.code
; Initialize the DS, ES and SS segments (as well as fixing SP)
mov dx, DGROUP ; set DS & ES to the start of DGROUP
mov ds, dx
mov es, dx
cld ; ensure that DIRECTION = FORWARD

; set the numeric field width to 3 characters
mov al, 3
call dec_set_width

; ask the user for two decimal numbers separated by a space
input_loop:
mov dx, offset Prompt ; print the prompt
mov ah, 9
int 21h
mov dx, offset inpBuf ; get the user's response
mov ah, 10
int 21h

```

```

; terminate the input buffer
mov di, offset inpBuf+2 ; point to user's response
mov bl, inpBuf+1 ; terminate the input buffer
xor bh, bh
mov byte ptr [bx+di], 0

; check for the QUIT response
mov al, [di] ; get the first character
cmp al, 'X' ; if it's an 'x' or an 'X' then quit
jz quit
cmp al, 'x'
jz quit

; convert the first number to binary
IFDEF SIGNED
call dec_read ; convert signed string to binary
ELSE
call udec_read ; convert unsigned string to binary
ENDIF
jc input_error ; was the input OK?
mov bx, ax ; store the number

; skip over the spaces
skip_space:
mov al, [di] ; Get the character
inc di ; point to the next one
cmp al, ' ' ; is it a space?
jz skip_space ; yep -- go check the next one
dec di ; oops! went one too far

; convert the second number to binary
ifndef SIGNED
call dec_read
else
call udec_read
endif
jc input_error

; print the results of addition
add bx, ax ; Add the numbers
IFDEF SIGNED
jo add_error
ELSE
jc add_error
ENDIF
mov di, offset outBuf ; format them into the output buffer
mov ax, 0d0ah ; put return/line feed into buffer
stosw
mov ax, bx
IFDEF SIGNED
call dec_format ; format signed number into buffer
ELSE
call udec_format ; format unsigned number into buffer
ENDIF
mov ax, 0d0ah ; put return/line feed into buffer
stosw
mov byte ptr [di], '$' ; terminate the output buffer
mov dx, offset outBuf ; print the results
mov ah, 9
int 21h
jmp input_loop

quit:
mov ax, 4c00h ; Return to DOS with ERRORLEVEL=0
int 21h

; There was an input error -- print the appropriate error message
input_error:
mov dx, offset TooLarge ; Either the number was too large
cmp ax, err$BADCHR ; or there was a bad character in
jnz ie_skip ; the input string
mov dx, offset BadChar
ie_skip:
mov ah, 9
int 21h
jmp input_loop

; The addition overflowed
add_error:
mov dx, offset Overflow
mov ah, 9
int 21h
jmp input_loop

end

```


OTHER JOURNALS

If you prefer...

You can replace your subscription to *Inside Assembler* with one of the journals listed below. Just call our Customer Relations Department at (800) 223-8720 or (502) 491-1900 to let us know which journal you'd like to receive.

Spreadsheet:

Symphony User's Journal
1-2-3 User's Journal (Lotus® 1-2-3® up to Release 2.3)
Inside 1-2-3 Release 3 (Lotus 1-2-3 Release 3.1)
The Expert (Microsoft® Excel - Windows™)
Inside Quattro Pro
Excellence (Microsoft Excel - Macintosh®)

Word Processing:

Inside WordPerfect®
Word for Word (Microsoft Word - PC DOS)
The Inside Word (Microsoft Word Release 5.5 - PC DOS)
Inside Word (Microsoft Word - Macintosh)
Inside Word for Windows

Systems:

Inside Microsoft Windows
Inside DOS (entry level/intermediate articles)

Programming:

Inside Turbo Pascal
Inside Turbo C++
Inside Microsoft C
Inside Visual Basic™
Inside QuickBASIC
Inside Microsoft Basic
Inside HyperCard®

Database:

Paradox® User's Journal
Paradox Developer's Journal (PAL Techniques)

Integrated Software:

The Workshop (Microsoft Works - PC DOS)
Inside Microsoft Works (Macintosh)

Graphics:

Inside Freelance (DOS)

*Important
for subscribers m.*

**LAST
ISSUE!**